ClearCase Configuration Management
Task Environment User's Guide
(For SCDO Only)

July 29, 1996
Revision 1: April 2, 1997
Revision 2: April 23, 1997
Major Revision 3: September 22, 1998
Revision 3.1: December 31, 1998

Prepared for: Raytheon Systems Company
1616 McCormick Drive
Landover, MD 20785

Ken Uhl
ClearCase/CM Consultant
kruhl@erols.com

TABLE OF CONTENTS

Setting Up the ClearCase Development Environment

The ClearCase "task" environment is intended to simplify branching structures within the development VOBs, automate developers' config spec manipulation, and capture essential defect tracking information for project status and analysis. By performing a few simple "initialization" steps, and then using the few scripts described below, new users can take advantage of this environment immediately.

This latest version of the "task environment" (which will be referred to as "version 3") has several significant changes from the previous releases. To review those changes, see Appendix A, "See What's New," of this document.

To use the environment, each user must first edit his local ".cshrc" to include the following lines:

setenv SUBSYS /ecs/formal/XXX (where XXX is you subsystem acronym)
source /tools/sde/config/cshrc

This will properly initialize your environment with the necessary environment variables and aliases. (NOTE: There is a known limitation on the size of variables within "csh," which may sometimes affect the "PATH" environment variable in particular. If after making the above changes, an attempt at "source"-ing the above "cshrc" file results in a "Word too long" or similar error, you have likely hit this limitation. To correct this, you will have to reduce the number of search paths contained within this variable – it is likely that some paths are not necessary for everyday work activities).

Once your environment is properly set, you will be able to use any of the following set of commands, which execute shell scripts:

cnewtask - to create a new task
csettask - to set to a particular task
clstask - to list available tasks
cpwt (pwt) - to determine the currently set task
cmergetask - to merge task changes (and, by default, submit them) to an integration branch
csubmittask - to officially submit (checkin) changes to an integration branch
cendtask - to end a task that has been successfully completed and merged
clscset (clspickup) - to list all of the merged change sets not yet integrated into the "current" build label
cpickupcset (cpickuptask) - to add a recently merged change set to your config spec
cdropcset (cdroptask) - to remove a "picked-up" change set from your config spec
csetinfo - to obtain detailed information about a particular change-set.
ctaskinfo - to obtain detailed information about a particular task.
cfixinfo - to obtain a list of change-sets included in a particular integration branch, release label, or to fix a particular defect.

You will use the standard clearcase commands - such as "cleartool checkin" and "cleartool uncheckout" - for other Clearcase functions that are not encapsulated in scripts.

The following sections provide an overview to understanding and using the task environment and concludes with a complete command reference.

Using the ClearCase Task Environment

The ClearCase task environment is based on a few fundamental concepts: these include the terms “task,” “baselabel,” “integration branch,” “primary integration branch,” and “change set.” The following sections briefly define and describe these concepts, then provide general usage information.

Concepts and terminology

Task

A “task” is simply any development or maintenance effort performed by a developer that involves editing a logically related set of files. A task is usually thought of in abstract terms such as work done for a specific enhancement request (capability ID) or an assigned defect identifier (NCR).

When put in more tangible terms, however, it refers to a particular ClearCase view and its corresponding set of config specs that are customized to each user and development goal. Thus, developers perform daily work on code within the context of a particular task or tasks, and such tasks are created and deleted as work on them begins and ends.

Baselabel

Tasks serve the essential function of isolating an individual’s (or small group’s) work from a known “stable” baseline until such time as that developer is ready to incorporate his/her changes with on-going group activity (i.e., for nightly or periodic builds). A “stable” baseline is considered one that has been (at least) successfully compiled, and perhaps has been tested or even released. Thus, the baseline essentially represents some “frozen” point in time that has some significance and that may serve as a basis or “anchor” point for future development or fixes. Such a baseline is typically defined in ClearCase by a single “labeled configuration” that is applied by the ClearCase Support Group. A useful term for combining the two concepts of “baseline” and “labeled configuration” is “baselabel.”

The baselabel is a key component in identifying where new work is to begin. It identifies the exact version of every ClearCase element that was used to build the named configuration. When used in a command such as “cnewtask,” it serves as the basis or “anchor” point for the sprouting of new task branches during checkout commands; when used for merging one branch to another, it serves as a basis from which the target (integration) branch of a merge should be sprouted.

Integration Branch

As tasks are completed, and the developer is satisfied with the changes on the task branch, (i.e., the code has been incrementally compiled and unit tested), those changes must be integrated (or “merged”) back into one (or more) “integration” branches for building and integration testing with the rest of the team’s changes. Thus, an “integration” branch is simply a branch that is used to “accept” changes from concurrent (parallel) task work. This merge process is generally referred to as “merging a task” and can be accomplished through use of the “cmergetask” script, which is described more thoroughly later.

Primary Integration Branch

All tasks are created (using "cnewtask") based on a known "baselabel" and targeted for eventual integration into a "primary integration branch." Whereas "integration branch" is simply a generic term for any branch that "accepts" merges from any other branch, a "primary" integration branch is differentiated because it has particular significance to a given task. A task's primary integration branch is defined at task-creation time (either by default through templates or as specifically directed by the user) and is closely related to the work being done in the task.

A task uses the primary integration branch as the place from which to determine whether newly branched checkouts have the latest code as submitted during the course of the day. In addition, the primary integration branch is used as the default "target" branch of a "cmergetask" command for non-quickfix tasks, and as the source of checkouts/checkins for quickfix tasks. Thus, when a user specifies a particular primary integration branch when creating a new task, that user is essentially declaring that the branch specified will be the default target of any subsequent cmergetasks.

The creation of a task will construct an appropriate view and a config spec specifically designed for checking out files to a user-specific task branch. This provides developers with the flexibility of working against a known baseline for most of the code, while at the same time allowing them to check out and in changes to their own unique task branches (to preserve and protect their incremental work) without incurring the danger of perturbing work being performed simultaneously by other developers checking in on other branches.

Throughout the course of the day, developers will make changes on their task branches, and (if authorized) merge those changes into the primary integration branch. Depending on the nature of a particular developer's work, that developer may or may not want to see other users' changes as soon as those changes are integrated to the "group's" integration branch. This model accounts for this in several ways.

Change Sets

By default, a task config spec will exclude changes introduced to the primary integration branch by other developers. This allow's each user to work in isolation until he decides he is ready to accept these new modifications. When desired, however, a developer has several ways to include the work of others.

The first is through a concept termed the "change-set." A change-set is simply a way of grouping a related set of code changes (very similar to the "pickup" labels previously used). The fact that a certain group of files was either worked on together or merged together (or more precisely, checked into the primary integration branch together) is what defines them as being "related" -- although in practice, the intention is that this "change-set" represents a single logical code change (versus a random set of changes that just happen to be worked together and checked-in together).

Change-sets can take two forms. The first is an "integrated change-set," which is a change-set that has been successfully integrated into an integration branch and identified with a specific change-set label. The second, a "non-integrated change-set," is identified by those files containing a change-set branch (which is simply the renamed task branch after the file has been submitted to the primary integration branch) and represents those changes made within the task prior to merging/submitting to the primary integration branch. If several "cmergetask/csubmittask" combinations are performed throughout the life of a task, the resulting change-sets are sequentially numbered for each merge/submission.

Thus, whenever developers checkin any files within a task, they are adding those files to a numbered "non-integrated" change-set associated with that task (unless they are in a "quickfix" task, in which case they are adding directly to an "integrated" change-set). Similarly, whenever such work is submitted to an integration branch, (by a developer performing a "cmergetask" [without the "-nci" option], or a "csubmittask") a new "integrated" change set is created and labeled within ClearCase. Other users can incorporate any integrated change-set into their own config spec by using the "cpickupcset" command (this works similarly to the old "cpickuptask" command, which is, in fact, now just an alias to "cpickupcset"), or drop previously included change-sets using "cdropcset" (again, the old "cdroptask" is now an alias to "cdropcset").

Including an integrated change-set label in one's config spec (as done by "cpickupcset") is generally only useful as a means to "picking up" those changes during the period (usually a day) prior to the group integration build (i.e., the nightly build); after that point, all change-sets will have become incorporated into the latest "current" build label, rendering the current change-sets unnecessary and potentially confusing. Therefore, once an official integration build has been performed for a particular integration branch, the system will "reset" each associated user's config spec by automatically dropping any picked-up change-set labels the next time a "csettask" is performed.

As an example, the following shows how one might use tasks and change-sets.

Suppose a developer is assigned to work on two distinct NCR's, one that has to do with a problem that was found in a particular GUI interface, and another that looks more involved and may require a database schema change. To address these problems the developer creates two separate tasks to keep work on the two problems isolated from each other (as two separate views and task branches are subsequently created). One task he calls "control_gui_fix" and the other he calls "db_schema."

The developer uses "cnewtask" to create the two tasks, then uses "csettask" to first begin work on the GUI problem. Discovering that the GUI problem is relatively minor, he makes three changes in two different files within his control_gui_fix task, then checks those changes back into his "control_gui_fix" task branch. Once he has compiled the changes and is satisfied with his functional tests, he then submits the changes to the "relb" branch using "cmergetask." Because cmergetask by default calls "csubmittask," he is prompted to supply the "Merge Form" identifier that authorizes this change to the primary integration branch.

Alternatively, if the developer did not want to submit the changes to the integration branch until he's had an opportunity to review the merge results and run a test compile, he may opt to use the "-nci" command with cmergetask which would leave the resulting merged files checked-out to his view. In this case, no one else would see the new versions until he was satisfied that everything re-compiled successfully -- only then would he issue a "csubmittask" to officially check into the integration branch those updated files.

Once "csubmittask" has been invoked (either through "cmergetask" or directly by the user), and the merge form identifier is verified, the user will be asked to supply an NCR number (or numbers) which logs this defect as being addressed (and potentially fixed) by this change-set submission. Upon successful submission of each of the files into the primary integration branch, a label will be attached to each file involved using a standard convention, (in this case, "CSET_relb.johndoe_control_gui_fix_1"). This label refers to the "integrated" change-set and serves to identify the versions of the files that were integrated into branch "relb" to fix this particular problem (included file versions can subsequently be listed via the "csetinfo" command).

Other developers then wanting to include this GUI fix with their own task work, can issue a "cpickupcset" to pick up those submitted changes. The raw changes (i.e., prior to the integration) will remain on the task branch that has been renamed "johndoe_control_gui_fix_CSET_1" to correspond with the newly submitted change set (and to allow new checkouts to continue using the original "johndoe_control_gui_fix" task branch). All files containing the renamed branch are included in the "non-integrated" change set for this fix.

Similarly, work on the "db_schema" problem would follow the same set of actions, only the developer in this case may decide to "integrate" this more complicated change over a period of time in the form of three or four separate submissions. Thus, instead of performing a single "cmergetask", he instead would perform a series of three separate "cmergetasks" over the course of two weeks, each being identified by a sequentially numbered change-set identifier. For instance, when he is finally finished with the task, the task may include change-sets "CSET_relb.johndoe_db_schema_1," "CSET_relb.johndoe_db_schema_2," and "CSET_relb.johndoe_db_schema_3."

The choice to partition the work into three sub-tasks was made to in order to create three relatively simple merges, versus waiting until the rest of the team's work got two weeks ahead and those changes would ultimately be much more difficult to incorporate with the latest baseline. (Note: According to this site's policy, however, for each of these three separate submissions, the developer must get prior approval through the "merge form" process).

Conversely, if the changes to the database schema were such that no portion of it could be integrated until the entire change was complete, the developer would have no choice but to finish all work on the task branch and perform a single more time-consuming merge at the end of the two weeks.

The VERSION_ON_CHECKOUT Variable

A second way for developers to obtain newer versions of files without having to wait to pick up an entire change set is through use of the "VERSION_ON_CHECKOUT" environment variable.

This variable, which can be set to either "latest" or "view-selected," allows users to choose between checking-out the "view-selected" version of the file "as-is" (so as not to introduce any unknown changes), or to instead obtain the "latest" changes from the primary integration branch that would not have otherwise been seen until merge time. This capability provides the added flexibility for developers to incrementally pick up the latest versions of checked-in files on a file-by-file basis, but only during the initial checkout time for each file. It does this without requiring that the user change the "static" config spec (selecting stable baselines), to a more "dynamic" config spec (with a "LATEST" rule), for *all* files – which would put "viewing" the latest versions out of the user's control, and may introduce too much confusion.

This flexibility is important for developers who want to maintain a general "stable" view for the majority of the files, but upon checking out a particular file, want to obtain the latest changes that may have been introduced over the course of the day (as they do not want to immediately start working on an out-of-date version).

Working on out-of-date versions happens routinely in this environment because views generally select particular "labeled" versions of all the files in the VOBs – in most cases such labeled versions are either some current build label, such as "CURRENT_BUILD," or some known baselabel such as "RELB_BASE." Because the config spec typically selects one of these labeled versions, an initial checkout will create a task branch from the labeled version, which is not necessarily the "latest" version on the primary integration branch. This behavior, however, is often desirable as it ensures that developers checkout exactly the same version of the file that was viewed just prior to the checkout, and furthermore, they are assured that this version is known to "work" with the rest of the configuration.

Nevertheless, a developer may also not want to work on a version of a file if he knows that version is already out of date – doing so would just require more effort later during the "merge" of that file to reconcile those changes. Instead, he may want to simply begin working on the latest version, making it work with his task-private files in advance, before submitting all changes back to the primary integration branch. If this is the case, the user now has the option of setting the VERSION_ON_CHECKOUT environment variable to "latest." Doing this will cause each checkout command to determine if there is a later version of the file on the primary integration branch, and if so, to incorporate those changes (via a merge) into the version being checked-out. This check is only performed on checkout's that result in a

newly created task branch – it is assumed that once work begins on the task branch, this branch should remain isolated from further updates from the primary integration branch.

Developers wanting to maintain the same behavior exhibited by earlier versions of this environment need only set the VERSION_ON_CHECKOUT variable to "view-selected" (or just leave the variable unset), and they will always branch from, and checkout, the exact version that was selected prior to the issuance of the checkout command regardless of whether or not it is the most recent version checked into the primary integration branch.

By default, on all checkouts, an information message will be displayed describing the mode chosen. This message can be suppressed by additionally setting the environment variable NOWARN_MERGEOUT to "1".

For additional details on this functionality, see the section "User-defined 'view-selected vs. Latest' checkout options" in Appendix A of this document.

Label promotions and the BUILD and STABLE labels

Each integration branch has associated with it a label (or labels) that are used to "roll-up" the current state of a particular development effort. These labels typically have the words "CURRENT," as in "CURRENT_BUILD," and "STABLE" as in "STABLE_BUILD," as part of their identifiers. These labels are moved, or "promoted," on a regular basis (usually nightly, in conjunction with the nightly builds). This not only allows regular integration builds to be performed at scheduled times, but provides users with a "fixed" baseline for making the following day (or days, if needed) changes.

Label promotions allow merged code changes to be visible to other developers at these regular intervals, as all developers typically have one or both of these "default" labels in the task config specs. By performing promotions/builds on a frequent basis, developers can not only take advantage of derived objects that will be "winked-in" from the regularly built views, or staged into a "staging" VOB, but can also begin their own view-private builds shortly after the nightly builds finish (as cron jobs in the early morning) to synchronize their views with the nightly build views and avoid costly recompiles during the daytime.

The "build" label is used to mark the "most recent" integration build and establishes a "daily" baseline of the most recent integrations. Before this label is "promoted" each day (or periodically) to the latest versions on the primary integration branch, the existing configuration is captured by a "placeholder" label referred to as the "stable" label. Thus, the "stable" label is simply a preservation of the previous "build" label. This method of promoting labels in this "lock-step" manner allows testers to verify the latest "build" results, while developers continue to work uninterrupted against the more "stable" baseline.

Thus, depending on whether an individual is working as a "tester" or as a "developer," he will have the environment variable "CLEAR_CSETS_ON" set to either keyword "BUILDLABEL" or "STABLELABEL" respectively. This variable is used by the scripts to determine which label (the exact label name will vary between projects) is being "watched" by the user (since the majority of users work as "developers," the default is set to "STABLELABEL").

The significance of the "CLEAR_CSETS_ON" label is that it provides an important mechanism for automatically updating config specs to coincide with the daily builds/promotions. Because developers may "pick up," or include, specific change sets that are integrated into the primary integration between build periods (see "cpickupcset" and "cdropcset"), there must be a mechanism to automatically clear these labels from these config specs once they become obsolete. (Certain change set labels will become obsolete once a nightly promotion has occurred because the "CURRENT" or "STABLE" labels may reside on versions that are more recent than the change set versions – hence, a config spec selecting an old change-set would not be viewing the most current versions).

The mechanism to clear the change set labels is built into the "csettask" command which determines whether or not the user's task has been set since the corresponding CLEAR_CSETS_ON label has been promoted – if the current invocation of the csettask is the first one since the label promotion, the script will automatically clear the out-dated change set labels from the config spec. In addition, the same mechanism will update the "created_since" rule to match the date and time of the last label promotion. This allows "nightly" builds and promotions to delayed for several days without adversely affecting which versions are being selected.

Daily task usage

Tasks are intended for use as short term activities. Thus, instead of developing for weeks or months at a time without merging code to the primary integration branch (using the script "cmergetask"), developers should be in the habit of merging as frequently as possible without breaking the primary integration branch. Frequent merges benefit the developer because they result in smaller and more automatic merges. (I.e., if a task is worked too long in isolation, the eventual merge to the integration branch will be larger, more complicated, and require much more conflict resolution amongst developers).

This all-important rule of "merging soon and often" not only applies to individual developer's private task branches, but even more so to larger "development branches" (such as "maintenance" or "new feature" branches) that must eventually be merged to some other main-line development branch. Performing incremental merges on a routine schedule (say, bi-weekly or monthly) will not only eliminate the horror scenario of a "massive merge session" that could last more than a week before the code stabilizes, but will increase the overall predictability of incorporating changes and meeting schedule deadlines.

As an example, development of a new interface involving an inter-process communication (IPC) between two subsystems should be broken down into smaller sub-tasks, such as 1) stub out IPC proxies, 2) develop code to send empty messages, 3) complete messages, 4) integrate IPC proxies into the code, and 5) bug-fixes. At the end of each of these 5 sub-tasks, a merge should be performed to integrate the code back into the primary integration branch. Each of these merges will be recorded as it's own uniquely named and numbered change set, which can later be queried to identify the files involved and any defects fixed.

It should be emphasized, however, that no code should be merged that will result in build failures of the primary integration branch. Thus, it is useful to take advantage of the "-nci" option in "cmergetask," which will perform the required merges but leave the files in a checked-out state. This allows developers to compile and test the newly merged code prior to its official submission using a subsequent "csubmittask."

General Usage Guidelines

There are several "types" of tasks that can be used in this environment. The following types of tasks *should only be used* in the situations described:

1. Maintenance Tasks
   For use in performing a patches to a delivered baseline.

2. QuickFix Tasks
   Only used to make fixes directly to the integration branch under the direct supervision of ClearCase Support Group team members. Typically used to correct compilation errors in the nightly build.

3. Shared Tasks

Only used to support interface changes between small groups of developers. Intended to allow interface changes to be visible between owners of the interfaces before the interface change is merged into an integration branch.

4. Private Tasks:
   For general use in all other cases. This is the most common type of task used in the ClearCase environment.

Creating a new task

To begin work in the task environment, developers must first create a task. The scripts are designed to simplify view and config spec creation. Creating a new task (see "cnewtask" below) first creates a new ClearCase "view." Additionally, based on options passed, will set up the appropriate config specs for that task (it does this by using standard config spec templates, customizing them to the user's account). This works to free developers from the chore of creating and manipulating config specs that accurately map to the appropriate branching structure for a given task. Finally, the "cnewtask" script will "setview" to the newly created task (or use "csettask" to switch between tasks) from where the user can simply change directory to any VOB and begin working on the files.

By default, a new task will be created that will allow developers to work off a developer private "task branch" from a pre-determined baselabel. Working on a separate task branch provides the benefit of allowing the developer to check in and out of his own private branch until he is ready to submit those changes into one of the nightly (or periodic) builds. It has the slight disadvantage, however, of requiring that the user perform a "cmergetask" to move changes back into an integration branch (depending on how long the period is between performing "cmergetasks" this may or may not be a trivial effort).

User's preferring the speed of working on code directly off the primary development branch may provide the "-quickfix" option when creating a new task. Despite it's name, however, this may not be the fastest method if a number of files are involved; the disadvantage of using "-quickfix" is that in order to check any file back into the primary integration branch, the user must provide a valid defect number (that has been approved for integration) with each checkin (instead of providing this just once during a "cmergetask"), or, alternatively, may check in all files together using "csubmittask." In addition, the quickfix does not allow for incremental "checkpointing" of changes prior to integrating the entire change (as a checkin immediately causes changes to be seen by everyone). In practice, the "-quickfix" option is best used only to fix errors that must be fixed immediately because the problem is preventing many developers from completing work.

Using tasks

Once a task has been created, developers can set to a task, or switch between tasks, using the "csettask *task*" command. This not only performs a "cleartool setview" to the specified task, but handles other housekeeping chores such as checking to determine if the master project template has been updated -- or if a significant build label has been promoted -- since the last csettask was performed (and updating the user's config spec accordingly). For example, a developer working on "new development for B1" while at the same time working on "NCR #123 for REL_A2" can switch back and forth between the two using "csettask newdev_b1" and "csettask NCR123".

A "csettask" is automatically invoked if the user performs a "cleartool setview" on a view-tag that was created as a task (i.e., using "cnewtask").

Integrating tasks

Integration of task work should be done as frequently as possible to avoid time-consuming and difficult merges that will occur when development diverges from the baseline for long periods of time.

If a default task was created (causing changes to be placed on a task branch) developers must periodically incorporate these changes with the rest of the group by performing a "cmergetask" to merge the changes back into an integration branch. The cmergetask will consider all files that have changed for the current task, and attempt to perform an automatic merge to bring those changes back into the baseline. If there are conflicts between two or more users' changes the xmergetool GUI will be automatically invoked to assist in resolving the conflicts.

By default, "cmergetask" will perform the merges, then automatically check in the results of the merge to the integration branch. Because this does not allow the developer to review or verify (i.e., recompile) these newly merged changes prior to their submission to the integration branch, using the default is ***not*** the recommended approach (although it was left this way for compatibility to earlier releases). Instead, the preferred method is to take advantage of the newly introduced "-nci" option, which will perform the same merge, but will leave the resulting files in a checked out state. Since these files are seen through the view that has them checked-out, the user can then re-compile the merged files, unit test them to make sure everything works as expected, and finally submit them officially to the baseline using the new "csubmittask" command.

In addition, if the "-nci" option was used, a developer wanting to "back out" of a merge can now execute the command "cmergetask -unco" which will revert all merged files (*that have not yet been "submitted" i.e., checked-in*) back to their original state prior to the initial cmergetask. (Note: If the "-tobranch" option was used in the initial cmergetask invocation, it must also be specified with the "-unco" option).

NOTE: See section "Special merges using 'cmergetask'" for performing double merges, shared task merges, and quickfix merges.

Ending tasks

Once a task is completed, *it is important* to clean up the task environment using the "cendtask" command. Use of this command at the completion of tasks will decrease the number of views throughout the network and ***greatly*** improve all-around system and build performances. (Build performance gains result from reducing the number of derived object candidates -- some of which may be old, stale, and worst of all, unusable -- that are considered by clearmake's wink-in "shopping" algorithm. The "cendtask" command will remove the view and its associated config specs, and, if the "-rm" option is used, will also clean up old change set labels and branches that are no longer necessary in the administrative VOB. In general, views should be destroyed with this command at least once every 2-3 weeks.

Failure to clean up views this way will eventually cause the periodic "view clean-up" script run by the ClearCase Support Group to detect views that have reached an aging threshold, and will warn user's several days in advance to remove it. If the offending views have not been removed by the deadline provided, a script will automatically delete such views, potentially causing the loss of important work (i.e., any checked-out or other view-private files).

Nightly Builds

Nightly builds are managed by a user account set-up specifically for this purpose for each subsystem. A standard view is used and a cron job is set up to perform appropriate label promotions and group builds. All tasks are automatically set to view new components created by the nightly builds, and any existing "picked up" change-sets (which become obsolete upon certain label promotions) will be dropped with the next use of a given task. In addition, the "created_since" rule in user's config specs will be updated to reflect the date and time of the last label promotion.

The exact time that the nightly build label promotions will be logged. This information is then used by the csettask command (upon its next use by the user) to update the "created_since" rule in the user's task config

spec. The adjustment is made depending on the value of the user's "CLEAR_CSETS_ON" environment variable. Depending upon the value (usually "BUILDLABEL" or "STABLELABEL") the task's config spec will be adjusted to drop any existing (now outdated) CSET labels, as well as changing the "created_since" time to match the time that the corresponding label was promoted. This effectively keeps the user's config spec synchronized with all work that has been integrated into the baseline.

Users may elect to set up their own cron jobs to run nightly builds for a particular task that they may want to work on in the morning (obviously setting up the private build to occur sometime after the group build is complete).

New User Scenario

A new developer, Julie Uhlie, has come on board and is immediately assigned an NCR (ECSed2001) that her manager feels should not be too difficult and will provide a good introduction in working within the task environment.

After reading the problem description for ECSed2001, she sets out to fix the defect.

She logs onto her account "juhlie," and after reading this scenario about herself, types the command:

```
# cnewtask –h
```

to get an idea of the usage syntax for creating a new task. It reports:

```
Usage: newtask [-shared | -quickfix] [-base baselabel] [-intg intgbranch]
       [-build buildlabel] [-stable stablelabel]
       [-code cs_code] [-template cs_template] task
```

She's immediately dismayed to see that there are so many options to choose from, and she doesn't remember (even after intently studying the on-line user's guide (at http://pete.hitc.com/scdocm/ccs/html/sde/userguide.doc)) what they all mean. She then executes:

```
# cnewtask –hh
```

to get a more thorough explanation of all the the options.

She relaxes a little realizing that, except for providing a "task" name, all the parameters are optional overrides to the default project template. She knows that she is working on the "scdo_relb" project, and her manager told her that this defect was to be integrated directly back to the "relb" branch, and should use the standard "RELB0_BASE" that most developers in her group use. This indicates that she can use all the standard defaults for her current project.

With a little more confidence, she decides to create a task using all the defaults. Feeling rather creative, she invents a name for her task, and calls it "ECSed2001":

```
# cnewtask ECSed2001
```

This generates lots of output, but she ignores most of it figuring it must mean something to someone. It does. It finally completes with the last two lines being:

```
…
Set task:   ECSed2001
Set view:  juhlie_ECSed2001
```

That's encouraging, but she's not sure she believes it really worked, so she decides to check her current task:

```
# pwt
Set task:  ECSed2001
```

That looks right. But so far she's just been using these unfamiliar scripts, so she wants to double-check using some "real" ClearCase commands she remembered from her last job. She executes:

```
# cleartool pwv
Working directory view: ** NONE **
Set view: juhlie_ECSed2001
```

"Great! It worked!" Now she wants to check the config spec to see what kind of rules she'll be using to look at her files. She remembers that the "cleartool" command has been conveniently aliased to the letter "c," and runs:

```
# c catcs
```

```
############################################################
# WARNING: AUTOMATICALLY GENERATED CONFIG SPEC
#           *** DO NOT EDIT ****
#    This config spec was automatically generated using
#    cnewtask or csettask and contains keywords
#    required by certain scripts.  Changing the config
#    spec may have adverse effects when using your task
#    environment.
############################################################
#
# CSCODES=v3,
# TASK=ECSed2001
# TASKNAME=juhlie_ECSed2001
# TASKBRANCH=juhlie_ECSed2001
# INTGBRANCH=relb
# BASELABEL=RELB0_BASE
# BUILDLABEL=STABLE_BUILD
# STABLELABEL=STABLE_BUILD
############################################################

element * CHECKEDOUT
element * .../juhlie_ECSed2001/LATEST
element * .../relb/{created_by(juhlie)&&created_since(27-Aug.19:58)} -mkbranch juhlie_ECSed2001

mkbranch juhlie_ECSed2001
element * STABLE_BUILD
element * RELB0_BASE
element * DISCOVER_STABLE
element * /main/LATEST
end mkbranch kuhl_ECSed2001


############# CSPEC_TEMPLATE=scdo_relb.cs ######################
```

"Hey, that looks pretty clean." She thinks, remembering some of the thirty-line config specs that her old team used to have to create to get the right files. "I may even be able to figure out what this one's doing." For the time being, she ignores all the comments and keywords at the top, and focuses just on the rules.

"Let's see, I've got my obvious 'CHECKEDOUT' rule of course for my checked-out files, and similarly, a rule to show my latest changes on the 'juhlie_ECSed2001' branch – looks like it just prepended my login

name to my made-up task name. That's good, no confusion here if someone else chooses the same name by some strange coincidence!"

"Now, I don't know what all this 'created_by and created_since' stuff is about -- I'll have to check on that later. Looks like there is a separate section grouped together for some reason around a "mkbranch…end mkbranch" clause – that must mean everything within the clause will be creating a new taskbranch on any checkout I might do. That makes sense – all my checkouts go to my personal task branch."

She recognizes the RELB0_BASE label, realizing it represents the common baseline for all her code changes, but she's not sure about the STABLE_BUILD label above and the DISCOVER_STABLE label below. She re-reads the task user manual, and discovers that "STABLE_BUILD" is simply the "floating" label that is moved (almost) every night with the nightly builds. This means that the "latest" officially built code must take precedence over the rest of the rules below.

Not sure of what the DISCOVER_STABLE label represents, she asks one of her co-workers, Jim, who has taken the "Discover" training course. Jim informs her that this label exists only in the Discover model build VOB (so it has no significance within the source code VOBs). The "Discover VOB," as it is called, primarily contains the results (psets and pmods) of a Discover analysis model build. She is further informed that the DISCOVER_STABLE label represents the last successful model build that has been verified by the Discover tools group. If she will be doing any Discover analysis as part of her development, this label will give her the most current official model build results from Discover.

Since Jim seems to know so much, she decides to press him further for an explanation of the "created_by && created_since" rule. He explains that this rule works in conjunction with the nightly promotions/builds and is directly tied to the "build" label (i.e., the one above the baselabel in her config spec). The "created_by" rule is, of course, assigned to the owner of the task, and the "created_since" rule reflects the exact time when the "buildlabel" was last promoted. Thus, this rule works to effectively allow each user to only see their own submissions to the given task's primary integration branch during the course of the day (or more precisely, over the period between builds/promotions). This is desirable because a user would want to specifically view files that he had just merged, but would not necessarily want to pick up other user's merges until he was ready for those changes (i.e., at some known interval, such as the nightly builds).

Content with these explanations, Julie returns to her desk and is ready to complete her assignment. At this point, she wants to get an idea of where all the source code resides, so she types a familiar command:

```
# c lsvob
```

which produces a listing of all VOBs for the project.

Still set to her task, and as she is part of the "MSS" group, she changes directory to the MSS code base:

```
# cd /ecs/formal/MSS
```

She spots a couple of suspicious files, taking a look at them with the standard UNIX "more" command, and brings a few up in her favorite editor.

She sees the problem immediately (a variable was not initialized), and makes a quick editing change to correct the problem. She attempts to save the file, but get's a "read-only" error reported from her editor.

"Oh, that's right – I forgot to check the file out!" she remembers. So she saves the file with a temporary name "file.c.temp" and exits her editor. From the command line, she issues the standard cleartool checkout command:

```
# c co –nc  file.c
```

The checkout, however, reports some kind of message that she has never seen before:

Automatically created branch type "juhlie_ECSed2001" from global definition in VOB "/ecs/ccs".

Created branch "juhlie_ECSed2001" from "file.c" version "/main/2".

##
Reminder: To obtain the latest code from the "relb" branch
into checked-out versions off newly created task branches --
set environment variable VERSION_ON_CHECKOUT=latest .

[TO SUPPRESS this message, set env var NOWARN_MERGEOUT=1]
##

Checked out "file.c" from version "/main/juhlie_ECSed2001/0".

That's right, she forgot about the VERSION_ON_CHECKOUT variable and what it means, but with Jim's explanation of the "created_by && created_since" rule, it seems to make a little more sense. She re-reads the section in the task user's guide about the VERSION_ON_CHECKOUT rule, and it becomes crystal clear!

Since the file is now successfully checked out, she copies her temporary file in place of the now-writeable version, and then checks it back in:

```
# cp -f  file.c.temp file.c
# c ci -c "Initialized variable init_this to 1." file.c
```

Her file is now successfully checked in. She decides that she should have put some comments in the code itself, so she checks out the file again, adds the comments, then checks it back in. She's now made version 2 of the file on her personal "juhlie_ECSed2001" task branch. She further notices that this variable has not been initialized in several other similar files, so she proceeds to make the same change in four other files. She checks out each appropriately, makes the changes, then checks them back in to her task branch.

Finally she's ready to make this change visible to the rest of the project. Before she does this, however, she wants to check to make sure which files will be merged. She executes:

```
# cmergetask -print
```

which reports on (almost) all files that will need to be merged. (If she changed the contents of any directories – that is, added some new files, deleted others, changed file names – such changes are not known to this command until the merge has actually taken place).

Satisfied that the list of files includes all those that she changed, she re-issues the cmergetask command. Since she'd like to verify the results of the merge prior to actually checking them in, she prefers to take advantage of the "-nci" option that will keep her files in a checked-out state:

```
# cmergetask  -nci
```

This performs the required merges. Although most of the files complete cleanly as "automatic" merges, she is surprised to find that a "same-line" conflict in one of the files causes a graphical merge tool to appear. The tool has taken her directly to the line in question, and highlights (in yellow) the source code line in the middle panel. (Sidebar: The panels are laid out such that the upper panel is an editing window which contains the results of the merge, the lower left panel is the ancestor version of the two files being merged, the lower middle and right panels contain the two versions being merged – with the taskbranch version generally falling in the middle).

The yellow highlighted source code line matches the yellow prompt at the top of the tool which asks "Accept Change?" Next to that prompt, she is offered the options of answering "Yes" or "No" to accepting the currently highlighted change. Since this first highlighted change refers to the edits she made, she resolves it by answering "Yes." The tool then takes her immediately to the "other side" of the conflict (the second versions panel), and asks her a second time if she also wants to accept the change. of the other user's edit to the same line(s). Since her changes are more appropriate, she answers "No" to this one, and the tool

completes a merge of the rest of the file. Finally, the last yellow prompt asks if she wants to save the results of the merge to a named file, she responds with "Yes," and the merge is complete.

Once the merge has been completed, she reviews the results by performing various "diffs" with previous versions of the files. In addition, she executes a "clearmake" to ensure that nothing she has merged causes a build failure. She has "clearmake –C gnu" command aliases to "cmake", so she issues that command.

Unfortunately, the make fails to compile everything. After some inspection, she determines that the failure was due to a header file that was recently checked in to the integration branch by someone that is causing an incompatibility to a "typedef" structure in her code. Since Julie had used the "-nci" option of cmergetask, she is free to check the offending header file – this one will not go on a taskbranch, as the "merge" config spec has taken over and now "sees" all the LATEST code on her integration branch. She might decide at this time that the entire merge was not good, and she'd rather back out of it and start from scratch. She could do this by simply typing:

```
# cmergetask –unco
```

and all the files would be unchecked-out, and the merged files would be removed.

She chooses, however, to fix the problem with the other user's header file. She checks out the file, makes a change to the typedef, saves the file, and is now ready to compile again. This time the compilation works! She executes the code she has just compiled and does some unit testing to make sure the fix works as expected. It does!

Having satisfied herself that the code is now safe enough to make available to the rest of the group, Julie issues a the command:

```
# csubmittask
```

which gathers up all the merged/checked-out files for her integration branch, and checks them in.

Because this is a checkin to an integration branch, it is considered a "submission" and therefore requires that she enter a "Merge Form" approval number as well as any NCR's/CAPA's that this change addresses.

Additionally, because this is a "submission," the taskbranches on each of these files are renamed to identify this specific "change set" group of files. Such renaming allows her to view all the versions of the files that she just merged and checked into the integration branch (otherwise, her task config spec that is now reverted from its "merge" form to its original "default" task form, would STILL SELECT the pre-merged …/taskbranch/LATEST version!). At the same time, a special "CSET" label is applied to the checked in version on the integration branch. This label allows other users to pickup Julie's changes (through use of the "cpickupcset" command) if they urgently need to see those changes prior to the nightly promotion/build making them generally available to the rest of the group.

Happy that everything seemed to work for her first change, Julie goes to lunch. Upon returning, she notices she has an e-mail (from herself), that confirms to her and other interested parties, that the submission was successful! Her changes are now ready to be included in the next nightly build to be incorporated with other users changes and included in future integration tests.

The remainder of this document provides the specific commands, options, and descriptions for each of the new task environment scripts. All users should become familiar with these commands.

Special merges using 'cmergetask'

"Double" or multiple merges

One option that was dropped from cmergetask was the "-double" option for merging a single task into two separate integration branches. To perform any number of merges to multiple integration branches, developers should use the new options "-tobranch" and "-base" in cmergetask to specify a target integration branch and associated baselabel for a task branch merge.

It is important to realize that the default mechanism for cmergetask (which is now actually implemented at checkin time on the primary branch) necessarily changes the name of each merged task branch. This behavior is obviously not desired during a multiple merge, as the original task branch name is required for each separate merge. As a result, USER'S MUST MERGE THE PRIMARY INTEGRATION BRANCH (e.g., "relb") LAST to postpone this occurrence. Alternatively, to avoid all renaming of task branches, a user may explicitly specify the "-tobranch" option even with the *primary* integration branch, as specifying the "-tobranch" option is the mechanism that causes branch renaming to be suppressed.

As an example, to perform a double merge of task "jsmith_taskA" to both the primary "relb" integration branch as well as the "maint_DROP4PL" branch, the user would set to her task and issue the commands:

```
% cmergetask -tobranch maint_DROP4PL -base DROP4PL
  <… merge output …>
% cmergetask
  <… merge output …>
```

-OR-

```
% cmergetask -tobranch maint_DROP4PL -base DROP4PL
  <… merge output …>
% cmergetask -tobranch relb -base RELB_BASE
  <… merge output …>
```

Again, the difference between the two uses is that the first will cause a renaming of the task branches involved (desirable if work is to be continued in the task), and the second will leave the task branch names unchanged (in anticipation of a second or third cmergetask). In any case, the recommended approach is to perform the default merge LAST on the primary integration branch – this will result in the "correct" versions of files being selected if work on the task is to eventually continue.

Shared task merges

Shared task merges are performed exactly the same as normal merges. It is, however, important to understand the implications of performing a shared task merge.

A shared task is identical to a normal task with the singular exception that each user's task, rather than having the login id prepended to the beginning of the task name, the keyword "shared" is placed instead. This allows each member of the "shared" group to "see" each other's files as soon as those files are checked into that shared task branch -- allowing two or more developers to work closely with each other without having to wait for formal integrations to occur.

Despite sharing the same task branch name, however, each user still maintains his/her own task environment (view and associated config specs) for that task. This has a couple of important implications.

The first is that, since an individual's task "records" each VOB that the user has checked files into, only *those* VOBs in which the user has done work will be considered in the merge. Therefore, if a shared task is actually being worked in two separate VOBs, say CLS and IOS, and a particular user has only done work in

CLS, only changes done in the CLS VOB will be merged. It is important to note, however, that ALL changes in that VOB will be included in the merge, NOT just that individual user's. This is an intended behavior in order to keep changes associated with each VOB consistent. It does, however, provide a mechanism that allows developers working on the same task, but in different subsystems, a way to exclude merging files in subsystems that they may not be a member, and consequently, may not know how to merge correctly.

Quickfix merges

As mentioned previously, quickfix tasks will not generally need to be merged into the integration branch, as a simple checkin of a file will cause it to immediately become part of the baseline. Developers must therefore be careful to make sure all files are checked in when finished with a quickfix task.

In the event, however, that another developer has checked out, modified, and checked back in a version of the same file while the first developer has been working on the file, the first developer will need to perform a merge before ClearCase will allow the checkin to proceed. Although this merge can be performed manually, executing a "cmergetask" or "csubmittask" within a quickfix task will still detect this and prompt for an appropriate resolution. Thus, as with a normal task, proper completion of a quickfix task should also include an execution of either cmergetask or csubmittask to ensure that no changes are inadvertently missed.

COMMAND REFERENCE

NOTE: All commands in the task environment will report usage when given the "-h" argument. To obtain a more detailed usage description similar to this command reference, use the "-hh" (or "-help") option.

Most commands accept and expect an abbreviated form of the fully-qualified "taskname." This abbreviated form is represented by the word "task" in these reference pages. Thus, where used, "taskname" refers to the fully qualified "user_task" representation, and "task" by itself refers to the abbreviated form which assumes the "user" portion depending on the UID of the person executing the command.

NAME cnewtask

SYNOPSIS

cnewtask [-shared | -quickfix] [-base *baselabel*] [-intg *intgbranch*]
[-build *buildlabel*] [-stable *stablelabel*]
[-code *cs_code*] [-template *cs_template*] *task*

DESCRIPTION

Creates a new task, "user_task," for the current user, which is used to perform any official file modifications within the ClearCase environment. By default, it requires no options at all and will use defaults set up in a project task template, as determined by the project environment variables "PROJ" and "RELEASE."

- Creates a new view (using the script "mkview").
- Creates the appropriate config specs.
- Sets to the newly created view with the appropriate config spec for that task.
- Logs task creation.

OPTIONS AND ARGUMENTS

Default: Requires no options and creates a new development task named by convention *user_task* which is used to check files out and in to developer-private task branch. All default options are obtained from a project task template. Project default options may be overridden with the following options:

-maint <*keyword*>
Creates a new task environment for maintenance work based on the specified release keyword. A config spec will be generated for the newly created view that will result in branches being created (on checkout) from a particular release label (by convention, "*keyword*_BASE") onto a specific branch in the form "maint_*keyword*". This label and the branch represent the task's "baselabel" and "primary integration branch" respectively. This option is used to abbreviate the more generic form of "cnewtask" which is to explicitly specify the primary integration branch and baselabel through use of the "-intg" and "-base" options respectively.

-quickfix - Creates a quickfix task. This suppresses creation of a developer-private task branch for checkouts/checkins and results in all checkouts being performed directly off the primary integration branch. This type of task is best used when the changes are expected to be small and it is desirable to check out files directly on the integration branch to avoid performing a "cmergetask." Each checkin, however, since it will be to an integration

branch, will be prompted for both a merge form number as well as an NCR number(s) to record any fixes being made. (Alternatively, the commands "cmergetask" or "csubmittask" may be used to check in all checkouts at one time – these commands will only prompt a single time for the merge form number and defect identifiers).

-shared Creates a shared task. This creates a new task using branch "*shared_task*" instead of "*user_task*" that can be shared between two or more people working closely on the same task branch simultaneously. It should only be used for small groups working closely together. Each user will still have his own separate view, but will have a config spec that matches the rest of the "shared" groups. This option is useful for working on changes to interfaces, but has the disadvantage that each developer sees the other's changes as soon as files are checked in. This SHOULD NOT be used for larger groups desiring to work together for longer periods of time and wanting to delay submitting group changes to the official integration branch at some later point. Groups that want to work this way should put in a request to the ClearCase Support Group to become a separate "project," which will have it's own integration branch and separate project repository assigned. Such groups can then work in isolation (from the main development effort) until such time as they would like to integrate their changes.

-base *baselabel* Override the project default base label for this task. All task config spec templates should contain a "BASELABEL" definition -- if this is not the case, this option will NOT create one. (See section "Concepts and Terminology," sub-section "Baselabel" for description).

-intg *intgbranch* Override the project default *primary* integration branch for this task. This branch will become the *default* target of any subsequent "cmergetasks," and will also be used by the "VERSION_ON_CHECKOUT" function when determining if a newly created task branch checkout should include the latest changes from this branch (see sections "Primary Integration Branch," and "The VERSION_ON_CHECKOUT Variable").

-build *buildlabel* Override the project default build label for this task. If the project config spec template does not contain a BUILDLABEL definition, this option will NOT create one. (See section "Label promotions and the BUILD and STABLE labels").

-stable *stablelabel* Override the project default stable label for this task. If the project config spec template does not contain a STABLELABEL definition, this option will NOT create one. (See section "Label promotions and the BUILD and STABLE labels").

-code *cs_code* Add programmer-defined config spec code to task config spec. (Not generally intended for end-users).

-temp *cs_template*
Use template other than the default "$PROJ_$RELEASE" project template. Alternative template name must exist in the project "templates" directory.
(currently $SDETOOLS/sde3/config/cspecs/templates).

task Unique name assigned to this task which will be used in subsequent "taskname"- related scripts such as "csettask". Unless used with the "-shared" option, the full taskname will be created as "user_task," but most commands will use the abbreviated "task" handle.

EXAMPLES

Create a default project task:

cnewtask gui

NAME csettask

SYNOPSIS

csettask *task*

DESCRIPTION

Sets the current working task to the named task. It is used by developers to work on a tasks previously created with “cnewtask”.

This command will be run automatically by the “cleartool setview” command if the view being set to was originally created with “cnewtask.”

OPTIONS AND ARGUMENTS

task Task name (abbreviated) used to uniquely identify this task.
The full task name is typically prepended with users login ID to form a full "taskname" such as "user_task.” Most commands accept the abbreviated "task" name although some specifically call for the fully qualified "taskname" (usually when the owner of the task cannot be assumed).

NAME cmergetask

SYNOPSIS

cmergetask [-f] [-print] [-nci | -unco] [-tobranch *intgbranch*] [-base *baselabel*]

DESCRIPTION

Merges tasks created with “cnewtask.” Among other things, its primary function is to perform a “cleartool findmerge” against all files that were checked out to the currently set task. It does this by searching all VOBs used by this task (i.e., those VOBs in which a checkin for this task has occured) to determine which files have the current task branches, and of those, which versions require a merge to the *intgbranch*. The need for a merge of the file (or directory) is determined by comparing its …/taskbranch/LATEST version with either the …/*intgbranch*/LATEST, the “*baselabel*” version, or the /main/LATEST version, whichever is appropriate (and in that order). If no *intgbranch* branch exists for the file, (and either the *baselabel* version or /main/LATEST version is used in the comparison), an *intgbranch* branch is created during the merge process to accept the results of the merge. This is significant because it means that a file need not have an *intgbranch* prior to the cmergetask execution in order for the script to determine a need for the merge and to perform a subsequent merge onto that (newly created) integration branch.

By default, cmergetask needs no options. If executed without options it will attempt to merge the currently set task, and use the default “taskbranch,” “intgbranch,” and “baselabel” as defined for this particular task when it was originally created with “cnewtask.” In addition, using cmergetask with no options will have the intended side effect of renaming all task branches involved in the merge. Therefore, it is important to perform a “default” cmergetask LAST if it is known that these changes must be integrated into more than one integration branch. If, instead, the “-tobranch” option is used to direct the merge to a non-primary (secondary) integration branch, the

corresponding *baselabel* for that secondary branch should also be specified. This *baselabel* is typically, but not always, the *baselabel* normally used to base task changes for the secondary *intgbranch*, and often assumes the following convention:

Secondary branch: maint_<keyword>
Corresponding baselabel: <keyword>_BASE

For example, if a task has been created for the "relb" primary integration branch, using RELB0_BASE as its baselabel, performing a "cmergetask"

Version 3 of this environment has been updated to recover more gracefully from interruptions (so that it is "re-startable") and to handle error cases (particularly those caused by existing "reserved" checkouts) much more reliably.

NOTE: Tasks created with the -quickfix option of "cnewtask" do not generally require a merge, but in some cases such tasks can nevertheless take advantage of this command. (In some circumstances, for instance, a user working in a quickfix task may have chosen to check files out from a non-latest version, and thus would need to merge such changes with the latest version prior to submitting (checking-in) those files. This command handles such circumstances.)

OPTIONS AND ARGUMENTS

-f force (suppress warnings)

-print Don't do the merge, just print what would have been merged. (This may not return a complete list of everything that needs to be merged since it cannot know about new files that may have been created in the task unless it actually performs a merge on its parent directory – i.e., until a task directory is merged, the findmerge command will not be aware of any new files that would have been introduced by the merge itself).

-nci No checkin. Merge files, but don't check in (submit) to the integration branch. Useful for verifying the results of a merge prior to officially submitting the changes to the integration branch. (I.e., checked-out files are only visible to that view, and therefore changes can be made prior to final submission). Use "csubmittask" to checkin files after verifying correctness – use "cmergetask -unco" to cancel all checkouts and effectively "back out" of a merge that used the "-nci" option. Config specs using the "-nci" option will be left in a "merged" state – use "csubmittask" to complete the "merge" transaction and reset the config spec to its default "task" state.

-unco Uncheckout merged files. Will unco (cancel) a merge performed with the "-nci" option. If "-tobranch" is used with the "-nci" option, it must also be used with the "-unco" option.

-tobranch *intgbranch*
Merge task to a specified integration branch.
By default, the merge will target the primary integration branch.

Using this option will suppress the renaming of the the taskbranch, even if “intgbranch” is the default primary integration branch. (Useful if the task is to be merged to more than one intgbranch – i.e., a so-called “double merge,” -- and thus, renaming of the taskbranch is an undesirable side effect). If the “-tobranch” specified follows the “maint_<keyword>” naming convention, the “-base” option is not required.

-base *baselabel*

Change base label of target merge (typically used in conjunction with "-tobranch" to change both the target integration branch and base label).

NAME csubmittask

SYNOPSIS

csubmittask [-nci] [intgbranch]

DESCRIPTION

Officially submits task changes to an integration branch. This will checkin all files currently checked-out that are intended for integration into (by default) the primary integration branch, or (as specified) some other integration branch. In addition, it will prompt the user to determine if the submission closes some outstanding defect or enhancement, logging the response as appropriate.

OPTIONS AND ARGUMENTS

-nci No checkin. Submit task as complete without performing additional checkins. Typically only used for “quickfix” tasks where all files may have already been checked-in individually, and the user now wants to record this fix as “complete.”

intgbranch An integration branch to which changes are to be officially submitted. (Defaults to current task's primary INTGBRANCH, so is not typically needed).

NAME clstask

SYNOPSIS

clstask [-s] [-l] [-all] [-act] [-cmp] [-old] [taskname]

DESCRIPTION

Lists user tasks.

OPTIONS AND ARGUMENTS

-s Short. Lists current user’s abbreviated task names without heading info.

-l Long. Lists full name (non-abbreviated) of current user’s tasks.

-all Lists all tasks, not just those of the current user.

-act Active. Lists only active tasks.

-cmp Complete. Lists only completed tasks.

-old Old-style. Lists only old-style tasks.
(to be converted using "csettask" or "cnewtask").

taskname When given a fully-qualified taskname ("user_task")
will display the location of the task's repository.

NAME cendtask

SYNOPSIS

cendtask [-f] *task*

DESCRIPTION

Completes work on a task by removing the view created with “cnewtask” and cleaning up related config specs, history, and log files associated with the task. By default, the task’s config specs and history are moved to a “complete” state for later backup or recovery. This behavior, however, can be overridden with the “-rm” option.

Used by a developer when a task created with “cnewtask” is completed, and the workspace should be cleaned up. By default, a confirmation prompt is issued prior to performing any deletions.

OPTIONS AND ARGUMENTS

-f force (suppress confirmation warning)

task Abbreviated name identifying this particular user’s task.

NAME reopentask

SYNOPSIS

reopentask *task*

DESCRIPTION

This script reverses the effect of a "cendtask" command.
(Everything except re-creating the task's view -- do that
with a subsequent call to "csettask" which will prompt the
user to re-create the task view).

It essentially moves the given task from the project's
"complete" state directory to its "active" state so the user
can resume working on a task that was prematurely ended.
It should be immediately followed by execution of
"csettask task" by which the user will be prompted to
re-create the view that was deleted by the "cendtask."

NOTE: It will not recover any view-private files that
were lost from the "cendtask" command and can not
re-open any task that was permanently scrubbed from the
project's "complete" state. Tasks that were inadvertently

scrubbed can be recovered from backup tape by the CSG, however, since "view storage" directories are not backed up any checked-out or view-private files that were in the view at the time it was deleted are permanently lost.

NAME cpwt (pwt)

SYNOPSIS

cpwt [-s]

DESCRIPTION

Prints currently set task in the same format as the "cleartool pwv -set" command.

OPTIONS AND ARGUMENTS

-s Short. Prints without the "Set task: " prefix.

NAME clscset (clspickup)

SYNOPSIS

clscset [-proj] [-user] [-all]

DESCRIPTION

Provides a listing of integrated change-sets (change-sets that have been merged into an integration branch), but have not yet been promoted by the periodic (nightly) builds to a label visible by default to all developers.

OPTIONS AND ARGUMENTS

-proj Specifies a particular project. By default, the "$PROJ" and "$RELEASE" environment variables are used to determine the project. For example, project "scdo_relb" is defined by the combination of the PROJ=scdo and RELEASE=relb variables.

-user *username* Limits the list of integrated change sets to just those owned by user "username," where "username" can be a "partial" name of the user (e.g., "-user smith" will find both users "jsmith" and "rsmith").

-all Lists all historical integrated change sets associated with this project, not just those created since the last label promotion.

NAME cpickupcset (cpickuptask)

SYNOPSIS

cpickupcset *cset_label*

DESCRIPTION

Add's the named change-set label to the current task's config spec. Used by a developer when interested in seeing another developer's merged code from within the current view. This should only be done when the developer has determined that she will need the other developer's code in

order to build and test. In order to simply view another developer's changes in progress, use the command "cleartool setview". It is generally not advisable to pickup more than one change set at a time. This is because two or more change sets may have updated the same file – if this is the case, than only one of the file versions can be selected by the config spec, and this may result in incompatibilities.

NOTE:
All change-sets picked up over the course of some period (generally, one day), will be dropped automatically by the next invocation of "csettask" after each scheduled build/promotion has occurred (typically, this is the nightly build).

OPTIONS AND ARGUMENTS

cset_label - the change-set label attached to all versions of elements created by a merge/submit change-set operation. Change-sets currently available for inclusion can be listed using the "clscset" command.

NAME cdropcset (cdroptask)

SYNOPSIS

cdropcset [-all] *cset_label*

DESCRIPTION

Removes change-set label(s) (added using "cpickupcset") from the developer's current config spec. Used by a developer when it is no longer desirable to see within the current view another developer's integrated change-set(s).

OPTIONS AND ARGUMENTS

-all drop all change-set labels in current task's config spec.

cset_label - the change set label attached to all versions of elements created by a merge/submit change-set operation. The change-sets currently selected by the developer's config spec may viewed using the command "cleartool catcs".

NAME csetinfo

SYNOPSIS

csetinfo [-s] [-fix] *cset_id*

DESCRIPTION

Lists change-set information about the named change-set identifier. A full listing includes the change-set name, its type ("integrated" or "non-integrated"), any defects the change-set fixes (if of type "integrated"), and the list of file versions that comprise the change set.

OPTIONS AND ARGUMENTS

-s Short listing. Prints only the files included in the named change-set.

-fix Lists only the defect identifiers fixed by this change set.

cset_id - the change set identifier, of either type “integrated” or “non-integrated.” Change-sets currently selected by the developer’s config spec may viewed using the command “cleartool catcs”.

NAME ctaskinfo

SYNOPSIS

ctaskinfo [-s] *taskname* | *task*

DESCRIPTION

Lists task information about the named task. A full listing includes the fully-qualified taskname, current status of the task, VOBs included in the task, and a list of both integrated and non-integrated change-sets that the task has created.

OPTIONS AND ARGUMENTS

-s Short listing. Prints only the taskname, status, and included VOBs of the task.

taskname | *task _id* - Fully-qualified or abbreviated task name.

NAME cfixinfo

SYNOPSIS

cfixinfo [-s] < -*intg intgbranch* | -*base baselabel* | *defect_id* >

DESCRIPTION

Lists all change-sets included:

A) In an entire integration branch
B) In an entire baselabel (release)
C) To fix a particular defect

At least one of the above options must be specified.

OPTIONS AND ARGUMENTS

-s Short listing. Reports without header information.

-*intg intgbranch* - Any valid integration branch.

-*base baselabel* - Any valid baselabel (release or patch label).

defect_id - Any valid defect identifier.

APPENDIX A

See What's New

Overview

This appendix briefly describes the newly designed ClearCase "task-environment." It explains the differences between the old environment and this new one.

Why upgrade the existing environment?

The current task-based environment was originally designed to meet very specific requirements with certain goals as defined by the original target group, FOS.

These goals, in general, were designed to reduce the required ClearCase learning curve for new (and current) users by providing an environment that was easy to work with and created a "standard" methodology of use. Specifically, this involved creating an approach that would:

- Define a standard "branching" model, and "enforce" its implementation through automated scripts.
- Alleviate the need for users to understand and construct their own ClearCase "config specs".
- Alleviate the need for users to understand how to "merge" code changes within the tool.
- Alleviate the need for users to create and maintain ClearCase "views" for various work efforts.

The current task environment, therefore, was primarily designed around simplifying the end-user's "interface" with regards to the more complex aspects of the ClearCase product, allowing them to be more productive sooner.

The environment as designed, however, did little in the way of assisting management in terms of gathering information about *what* was being worked on, *why* it was being worked on, and *when* and *where* certain "logical" changes where integrated into baselines and releases.

Answers to some of these questions are currently being gathered to some degree through other systems such as DDTS (for defect tracking), and STTS (for CCB change requests and merge approvals). Although most software changes must be processed through these systems, no effort to this point has been made to integrate these (i.e., tracking and enforcement) within the day-to-day source code development framework.

Since the "task-environment" is currently in use by all developers, it provides an ideal foundation for developing such an integration in a way that will impact the developers as little as possible, yet yield additional vital data to management about the status and content of current and past releases.

In addition to these "management-level" requirements, use of the current "task-based" environment over the past year has revealed a few areas of limitations, some inefficiencies, and also some specific "bugs" that should be addressed in an upgrade effort.

First, in the category of "limitations," it was found that the current source code "branching model" was designed around some specific naming conventions and assumptions generated as a result of being written specifically for the FOS environment. This design has resulted in the environment not being as "generic" as it potentially could be – this will increasingly become a liability as the project moves from a "new code

development" environment to more of a "code maintenance" environment over the coming year. Specifically, greater variances of the branching model will need to be accounted for as the number of patch requests increases along with the number of baselines being actively maintained.

Next, in the area of "inefficiencies," the old task environment was designed around ClearCase 2.1. In the latest version of ClearCase (v3.1.1 -- to which this site has recently upgraded), there have been some notable improvements made to the product both in terms of improved performance as well as enhanced capabilities for administrative support. The new task environment can incorporate a lot of these changes to improve the overall efficiency of daily activities.

And finally, there were some specific "bugs," "limitations," and other requests for specific improvements, that were identified and which should, of course, be addressed.

Therefore, to make these adjustments and to enable the new capabilities, the next version of the task-based environment has been designed with the following additional goals:

- Retain the simplicity of the original task-based environment as much as possible.
- Fix known deficiencies.
- Provide more robust capabilities such as a "re-startable" mergetask, and appropriate handling of user interrupts.
- Remove restrictions that confine "task" use to a single VOB.
- Add features that will allow "logical" code changes to be tracked and managed between ClearCase and the other problem-tracking mechanisms already in place at the site.
- Add capabilities to identify and report on which releases particular code changes have been integrated.
- Introduce an enforcement mechanism to safeguard unapproved changes from being introduced into a baseline, yet will not interfere dramatically with current developer tasks.

The following section describes a design for implementing these goals.

New Implementation Design

In order to implement the goals defined in the preceding section, the following items were targeted for the new environment.

- Reduce the number and complexity of required config spec templates.
- Eliminate completely the "version 0" problem.
- Make "cmergetask" more robust – i.e., handle user-interrupts, eliminate "one-shot" completion requirement, and allow it to be "re-startable".
- Allow for non-template (i.e., command line argument) overrides of project template defaults.
- Allow tasks to span more than one VOB
- Incorporate a "change-set" model for tracking and managing related file changes.
- Integrate "change-sets" with existing defect-tracking and policy enforcement tools.

The following sub-sections describe the technical implementation of each of these new features.

Config Spec template simplification

Config spec templates are template files that are used by the "cnewtask" and "csettask" scripts to build a customized config spec instantiation for each users task view. This eliminates the need for developers to determine for themselves the needed baselines and configurations to work on a particular task – an often

confusing process – and instead allows them to begin working immediately on the problem at hand rather than the problem of setting up their environment correctly.

In its current implementation, these templates were designed with specific assumptions and therefore constraints on a specific branching model. By re-designing the config spec template to make it more generic, it is possible to reduce the number of required templates down to one -- and that one template can be simplified to less than 10 config spec rules. This not only simplifies the overall maintenance of the task environment, but is flexible enough to handle **ALL** future maintenance branching models.

Figures 1 and 2 below represent the new simplified config spec templates. Note, however, the ".mrg" *template* file will no longer be required in the "/tools/sde/config/cspecs/templates" directory, as it is generated directly from the cnewtask script based on the ".cs" template and/or user-passed parameters. An instantiated ".mrg" *config spec* file, however, will continue to be generated and placed alongside the instantiated ".cs" file for each specific task (in its own task repository).

The new templates will also have more appropriately named keywords such as “TaskBranch” and “IntgBranch” that more clearly describe their functions.

The new keywords are:

- **CSCODES** – These are special codes used by the template parser programmer and are generally not relevant to the end user. (But will be documented more thoroughly later).
- **TASK** – This is the “task” name exactly as passed in by the user.
- **TASKNAME** – This is the fully generated task name, usually prefixed with the user login id. It is used as a unique “task handle” (or task “tag”) to name the associated view, config spec, and task repository, and typically (but not always) task branches.
- **TASKBRANCH** – This is the name of the generated ClearCase branch type used where task-related code changes will be made. It is usually named identically to the TASKNAME, but sometimes (as in the case of a “shared” task) it has some other name.
- **INTGBRANCH** – This is the name of the “integration branch” – the branch where the TaskBranch will eventually first be merged onto (i.e., target branch of the “cmergetask”). A TaskBranch can be merged onto any desired branch, but usually the task was originally written for one particular development stream (i.e., release).
- **BASELABEL** – The label for the baseline from which the integration branch or the taskbranch will initially be sprouted.
- **BUILDLABEL** – The nightly (or periodic) build label, from which daily changes will be branched.
- **STABLELABEL** – Optional label for preserving the “last known good” configuration prior to updating the BUILDLABEL (i.e., it retains the previous BUILDLABEL).

Figure 1: Template “v3task.cs”

```
############################################################
# WARNING: DO NOT EDIT THESE COMMENT LINES
#       These comments may contain keywords required
#       by certain scripts.  Changing config spec comments
#       may have adverse effects in config spec usage!!
############################################################
#
#
# CSCODES=CSCodes
# TASK=TaskID
# TASKNAME=TaskName
# TASKBRANCH=TaskBranch
# INTGBRANCH=IntgBranch
# BASELABEL=BaseLabel
# BUILDLABEL=BuildLabel
# STABLELABEL=StableLabel
############################################################

element * CHECKEDOUT
element * .../TaskBranch/LATEST
element * .../IntgBranch/{created_by(USER)&&created_since(yesterday.05:00:00)} \
            -mkbranch TaskBranch


mkbranch TaskBranch
element * BuildLabel
element * StableLabel
element * BaseLabel
element * /main/LATEST
end mkbranch
```

Figure 2: Generated Config Spec “v3task.mrg”

```
############################################################
# WARNING: DO NOT EDIT THESE COMMENT LINES
#       These comments contain keywords required by
#       certain scripts.  Changing config spec comments
#       may have adverse effects in config spec usage!!
############################################################
#
#
# CSCODES=v3,merge
# TASK=TaskID
# TASKNAME=TaskName
# TASKBRANCH=TaskBranch
# INTGBRANCH=IntgBranch
# BASELABEL=BaseLabel
# BUILDLABEL=BuildLabel
# STABLELABEL=StableLabel
############################################################

element * CHECKEDOUT
element * .../IntgBranch/LATEST

mkbranch IntgBranch
element * BaseLabel
element * /main/LATEST
end mkbranch
```

Version “0” problem

In addition to creating more generic and flexible config specs, the config spec re-design along with some new features of ClearCase should virtually eliminate the “zero-version” problem at this site.

The “zero-version” problem occurs when a user checks out a file that results in several “cascading” branches being created. This is typically done to enforce a consistent branching scheme within the VOBs. Thus, in order to create several branches – one right after the other – the tool creates a branch from a previous branch, constructs a “zero-version” of the file (an exact copy of the branched-from version), then creates yet another branch from that zero-version. This continues for as many branches as the “config spec” has defined.

The problem this causes is that users with config specs set to look at the “latest” version on a particular branch will suddenly select this “zero-version” of a file if their view was selecting some other version (i.e., that file did not have that particular branch prior to the user’s checkout command). Although identical to its predecessor, this version would often cause “clearmake” to think it needed to rebuild any derived-objects that depended on the file (i.e., because of the new time-stamp on the file, it performed unnecessary and often time-consuming rebuilds).

This has been corrected by a combination of two changes. The first is that the new config spec design does not create a “cascading branch” situation at checkout time (when the intermediary branches are not really needed). Rather, the creation of any required intermediate branches (usually just one) are performed at “merge” time – the only period in which the existence of such branches is really required. Because “cmergetask” typically finishes its work and checks in new (and appropriate) changes to the integration branch in a matter of minutes (depending on the number of files), this significantly reduces the “window of

opportunity" that the zero-version files are selected. Thus, a rebuild will occur only when an "real" change has been made to the branch in question.

In addition, Rational engineers have corrected "clearmake" to recognize that any "zero-versions" that were identical to an earlier selected version do not need to be recompiled. This fix has been introduced with version 3.2 of ClearCase, and represents one of several significant improvements to clearmake.

Thus, the combination makes for fewer "zero-versions" to begin with, and those that *are* created, will no longer be problematic after 3.2.

User-defined "view-selected vs. Latest" checkout options

One significant aspect of the original FOS task environment that was not carried forward to the SCDO environment was the ability for the user to dictate which version of a file (currently being viewed on the "integration" branch) would be used as a basis for the checked-out file.

The current environment is set up such that the "view-selected" version of the file (on the integration branch) is the version from which a "task-branch" is created and checked-out from. This is often the preferred method because it provides the user with a more "static" environment. In other words, the "configuration" that the user is typically looking at is generally based on one or more "frozen" labels (versus a "LATEST" label that can be updated at any time by another user's checkin), and any changes that the user makes defines ONLY HIS DELTA changes from the known baseline. This situation effectively isolates the user from randomly "seeing" other user's changes before the first user is ready to integrate them.

While mostly advantageous, this is often, however, not the desired mode. Sometimes, while the developer still wants *the majority* of the files being viewed to be consistent with the "frozen" baseline, he also does not want to check out and work on a file which may not include the latest updates – it would mean he's essentially making changes to a file that has already had other changes made to it and checked into the baseline. A typical example in the SCDO environment would be users working on code in the morning that was based on the nightly build (CURRENT label). Over the course of the day, other developers may have been working on approved integrations to the baseline, which effectively updates the latest versions of files on the "integration" branch. Now, if some developer later in the afternoon decides to work on one of the files that has already been integrated, that second developer will *not* check out the latest version, but instead will branch from and checkout the labeled "CURRENT" version that was actually out-of-date by that afternoon. The current task environment provides no easy mechanism for the developer to even know about this condition of a given file, let alone adjust for it.

To rectify this problem, the new task environment provides for an environment variable setting that allows the developer to explicitly choose in which mode he will work. The environment variable, VERSION_ON_CHECKOUT, can be set to either "view-selected" or "latest" to reflect this desire.

This environment variable works in conjunction with a trigger script (attached as a post-event trigger to the "cleartool checkout" command) that will check to determine if the user is checking out the latest version of the file. If the file has been updated on the integration branch, and the user has specified "latest," the script will automatically include (i.e., merge) those changes into the version being checked out. This provides the user with the best of both worlds – he can maintain a static configuration for the majority of the files he's working with, yet still have the option of including the latest updates to files that he is checking out.

Template override options

Using templates to construct individual user's config specs has the important advantage that *all* config specs can be maintained and changed from one central location. If a branching structure or some common label name needs to be updated site-wise, a single change in the centralized template will cause ALL user's templates to be updated -- once the users re-set to the task -- to reflect the new changes.

The downside of this approach is that all users config specs will, as a result, remain synchronized with everybody else's at all times. This is prohibitive if tasks need to be created that are slight variations of the standard site tasks.

To address this issue, several command line options have been added to "cnewtask" to allow user's to "override" the site defaults. Thus, if the site template is updated, anything that was "overridden" by the user takes highest precedence and will not be updated with the rest of the config spec.

Tasks spanning multiple VOBs

Previously, tasks were limited in their use (at least automatically) to a single VOB. While this has not been a significant restriction at this site (as developers typically work in only one subsystem – and each subsystem is typically defined by a single VOB), with the release of ClearCase 3.0 this restriction need not exist.

The reason the restriction existed in the first place was that certain meta-data associated with tasks (i.e., labels and branches) were defined on a per-VOB basis. Since tasks were created based on the "SUBSYS" environment variable (which defines the primary VOB a user would be working in) only their subsystem VOB was initialized up-front with the appropriate label and branch type definitions (it is simply not practical to assume all VOBs will be used, and therefore initialize each accordingly). If the user needed to make task changes to some other VOB (say, COMMON), they would encounter errors on checkout attempts because they appropriate task branch did not exist. They would then either have to manually create the appropriate types themselves, or enlist the help of the CSG. In addition, if scripts such as "mergetask" were required to search every available VOB for potential merge candidates, the merge process would take up an exorbitant amount of time and resources.

With ClearCase 3.0 and later, there exists a new concept called an "administrative VOB," or "adminVOB." An adminVOB can be any normal VOB (but is usually a designated stand-alone VOB set up solely for this purpose) that is referenced by other VOBs in order to obtain "global" meta-data definitions if such definitions do not currently exist in the "local" VOB. It's implementation in ClearCase is fairly straightforward: a special "adminVOB" hyper-link is created by the administrator in any "local" VOB that may need access to "global" data. If an action in the local VOB (such as a "checkout" that must create a branch) does not find a local "instance" of, say, the branch type it needs to complete the operation, that operation will check for the existence of any defined "adminVOBs", and if found, attempt to obtain the needed branch type definition from its associated adminVOB(s).

The task environment at this site can take advantage of this new capability by first automatically defining all instances of branch types, label types, etc., in the central "admin VOB." Once that has occured, any operation in any VOB referencing that admin VOB can obtain the definitions it needs in real-time (on-the-fly), instead of requiring that they be predefined in any VOB that the user may ultimately need to access.

This capability is further augmented by a logging mechanism (see below section on "Defect tracking integration") that keeps track of, among other things, VOBs used in the course of a defined "task." This list can then be used to identify and limit the scope of work required by such scripts as "cmergetask."

Fixes to and the addition of a re-start capability for "cmergetask"

The script "cmergetask" handles the "merge" activity of "bringing-in" user tasks from their task-branches to the primary integration branch. This script is invoked when users have completed work on some logical unit of work, and are ready to integrate those changes with the rest of the development team.

While this script hides much of the complexity of this process from the end user, it has also been responsible for introducing some of its own confusion when portions of the merge operation fail or return unexpected results to the user. This sometimes occurs due to the user's lack of familiarity with the process, but also at times has resulted from deficiencies in the script itself.

The current version of this script is not as robust in recovering from error conditions as it could be. This situation is manifest most when the script successfully merges most of the files, but has undetected errors on a few files. Because the script did not detect those errors, it assumes valid completion, and as part of resetting an appropriate environment, renames the original task branch (this is done on purpose for reasons that will not be discussed here). Once the branch has been renamed, it is effectively hidden from the user's standard task, and thus, it becomes difficult for the user to correct the errors without the assistance of the ClearCase Support Group. Problems with "merging" have probably accounted for most of the problems reported to the CSG with regards to the task environment.

To correct this situation, the "cmergetask" script has been completely re-written. First, all known errors were, of course, eliminated. Secondly, the script reports much more informative information about what it's doing and any errors it has encountered along the way, and allows user's to interrupt the processing at any time to fix the problem. Finally, it has been changed in a way that allows for incomplete merges to be re-started and to pick up where the merge has left off, or conversely, to be backed out of completely.

Change-set model implementation

The addition of a "change-set" model to the task environment is probably the most significant aspect of the upgraded environment. This model provides a solid foundation on which to base data collection for management queries and decision-making, and provides a more logical integration with other defect tracking tools (see next section below) as opposed to the standard file-by-file integrations more typically implemented.

The concept of a "change-set" has been a highly requested (by ClearCase customers) improvement to the ClearCase product in general. Atria (now Rational) engineers responded, in part, to this request several years ago by including the concept of a "change-set" – as well as many other new enhancements – to an add-on package to ClearCase called "ClearGuide."

The ClearGuide product -- designed as an "out-of-the-box" process layer built on top of a ClearCase foundation -- was officially released early last year, and is compatible with versions 3.1.1 and higher of ClearCase.

Unfortunately, for those customers who cannot upgrade to a ClearGuide environment, little relief in this regard is offered through the native ClearCase environment. Therefore, to address this issue at this site, a basic implementation of change-set concept is being added to the "task-based" environment. Actually, this site already had the beginnings of a change-set model with its use of the "pickup" labels – in fact, the original "pickup" commands have been replaced (but aliased for compatibility) with equivalent "cset" commands. This concept, therefore, is not entirely new here.

Just what exactly are "change-sets" then? (and more importantly, why do we care about them?). Simply put, a "change-set" is a grouping, or set, of file changes that together comprise a fundamental "logical" change to the system. For example, "all file changes applied to the DROP4 baseline that were made to fix

NCR #1234" may be considered a complete change-set. Thus, the change set may be as simple as a single line change in a single file, or as complex as hundreds of lines of code changed in as many files. Those changes which are included in a given change-set are therefore at the sole discretion of the developer (who, generally speaking, is typically the only one who really cares which files need to be touched) -- but the point is that the "logical" change is being tracked as an entity itself, and the exact names and versions of files becomes ancillary (although still useful) data.

Why then, are change-sets important? Any site, particularly large development and maintenance operations such as this one, has a fundamental need to track and manage changes made to their software products. This need is most typically manifest in the most basic questions as "Which bug fixes and enhancement requests were included in the last release?," and "What files were modified as a result of fixing NCR#4321?"

While such questions are usually answered through the use of a "problem request" database, or other "defect tracking" system, if such systems are not integrated with the version control/configuration management system, the resulting reports may be incomplete or inaccurate, as they must rely on the accuracy of the person entering the data – assuming such data entry is even required, enforced, or performed.

While it is generally agreed that such an integration is a good idea in theory, it is often much more difficult to get consensus on how it should be implemented. The most common technique is to place a trigger on all checkins (and optionally, checkouts), that prompts the user for tracking numbers and subsequently verifies state information and permissions (this is in fact, basically how the ClearCase-DDTS delivered integration currently works). This is often overly intrusive to the user, slows down his interaction with the system, and can result in overall performance degradation for large sites such as this (i.e., it essentially forces the defect tracking system to become a central bottleneck for verification requests). The following section describes how the change-set model can be used in conjunction with the task-based environment to track and manage the necessary information without significantly altering user interaction with the system, and without slowing considerably their response times.

Reduced reliance on nightly builds (promotions)

In the current environment, if a nightly build is not performed (or more precisely, a promotion of the "CURRENT_BUILD" and "STABLE_BUILD" labels has not been performed), adjustments have to be made to the developers' config specs to see the work they checked in to the integration (relb) branch the day before. The reason for this is that there is a line in the config spec that contains a rule similar to:

```
{created_by(USER)&&created_since(yesterday.05:00:00)}
```

This rule exists so that developers can see code they merge over the course of the day that would otherwise be obscured by one of the other (CURRENT, STABLE, BASE) labeled versions.

This will be less of an issue with the new environment as the merging process (cmergetask/csubmittask) handles placing the newly created change-set label in the users config spec. This label will remain in the user's config spec until the next label promotion occurs, regardless of when that is. (Once the label promotion occurs, all config specs will be updated on the next issuance of a "csettask" to clean out the change set labels).

This process allows change sets that have been integrated over the course of, say a week, to remain visible even if the one-day "yesterday" rule has since expired.

Defect tracking integration

Defect tracking systems are designed to track and manage the reported, proposed, and approved changes to any system. They usually do not go as far as tracking the actual software changes made to the system itself – this is typically left to the version control/configuration management system. Although these systems are responsible for tracking different things, the need for an interrelationship between the two is obvious.

Because version control systems by nature track individual delta's to specific files, an obvious integration between the two systems is to associate each file change with a particular change number (either a NCR number or some "enhancement request" number). For this reason, the easiest implementation (particularly with ClearCase) is to place a "trigger" on every checkin/checkout command issued by the user and to verify and track those changes with the associated number in the defect tracking system.

Such an implementation, however, is often met with resistance because of its intrusiveness to the end-user, and its reliance on real-time verification of *every* change to the system. A more ideal solution is to balance the need to obtain this information with the need for the developers to get their work done as quickly as possible with minimal intrusion from the system. The combination of the existing "task-based" environment at this site, along with the underlying support of the "change-set" model, should provide such a compromise. Basically, it works like this:

Developers currently create "tasks" to work on some component of their subsystem. Check-ins and check-outs done within the context of a task are typically placed on a specific "task branch." Significant events within the task (such as checkins, checkouts, and mergetasks) are logged *against that task* in a separate repository. For performance reasons, since this site has already a considerable investment in NetApps fast file servers, this "repository" for logging (which alternatively could be done as meta-data in the ClearCase VOBs themselves, or in some other database including the defect tracking system's) is written directly to flat files on the NetApps.

Events associated with a task will be logged in a directory created solely for that task. Events associated with an integration baseline will be logged to a file that tracks approvals and changes to that baseline's designated "integration branch." Since changes to the baseline are the key changes that management cares about, it is much easier to track and approve checkin's to that branch than to get bogged down in everyday changes made on the task branches. Because the new environment "is aware" of "task branch" versus "integration branch" activities, it can appropriately adjust to the different types of source code changes.

The net result is that the user is normally free to work unencumbered for most daily activities (all the while, their events are being automatically and quietly logged appropriately), and only asked for "defect" validation during more significant events such as would occur during a "mergetask" operation.

The defect tracking interface from the user's perspective will simply be a prompt requesting the user to provide a (list of) defect numbers that a particular file checkin (or mergetask) to the "integration branch" will fix. This list is then validated against the defect tracking system prior to allowing the operation to proceed.

Behind the scenes, each file involved in such an "integration" will be both marked (with a label) and logged (in the integration log) as part of this "change-set" (see above discussion on change-sets). This type of identification can then later be used to quickly query a particular release for information about the specific change-sets, and hence, the specific file versions, that were incorporated into a given release.

In order to make this information available to the defect tracking system, "querying" utilities can be provided as an interface to the task-based environment, or periodic cron jobs can be established to directly update the defect tracking system with data gathered throughout a period of time (this eliminates the need

for the user to continuously wait for real-time updates between the two systems -- which is the more common approach).